electronics

MDPI

Article

ModDiff: Modularity Similarity-Based Malware Homologation Detection

Huaqi Sun, Hui Shu *, Fei Kang and Yan Guang

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China; mr_sunhuaqi@163.com (H.S.)
* Correspondence: shuhui123@126.com

Abstract: In recent years, the number and scale of malicious codes have grown exponentially, posing an increasing threat to cybersecurity. Hence, it is of great research value to quickly identify variants of malware and master their family information. Binary code similarity detection, as a key technique in reverse analysis, plays an indispensable role in malware analysis. However, most existing methods focus on similarity at the function or basic block level, ignoring the modular composition of malware. Implementing similarity detection among malware modules would greatly improve the efficiency and accuracy of homology detection. Inspired by the successful application of deep-learning techniques in program analysis, we propose a binary code module similarity detection method called ModDiff. It abstracts malware into attribute graphs, clusters functions using graph-embedded clustering algorithms to decompose malware into function-based modules, and calculates module similarity using graph-matching algorithms and natural language processing-based function similarity detection algorithms. The experimental results indicated that ModDiff improves the accuracy of module partitioning by 10.8% compared with previous work, and the highest F1 score of 89% is achieved in malware homologation detection. These results demonstrate the effectiveness of ModDiff in detecting and analyzing malware with important application value and development prospects.

Keywords: binary code; graph embedding; graph matching; modularization; similarity detection

Citation: Sun, H.; Shu, H.; Kang, F.; Guang, Y. ModDiff: Modularity Similarity-Based Malware Homologation Detection. *Electronics* **2023**, *12*, 2258. https://doi.org/10.3390/electronics12102258

Academic Editor: Aryya Gangopadhyay

Received: 13 April 2023
Revised: 11 May 2023
Accepted: 15 May 2023
Published: 16 May 2023

1. Introduction

Malware is one of the most prevalent security threats today, capable of stealing sensitive information, disrupting systems and networks, and causing significant damage to users and businesses. According to the AV-ATLAS Institute, the number of malwares has reached 1.253 billion as of April 2023, with over 252,000 new ones being added every day [1]. With such a staggering number of malwares, cyber security has become an increasingly pressing concern [2]. Malware homology detection is a crucial technology in this regard, as it can effectively identify malware families [3] and attacks [4] to avoid unnecessary losses to users and enterprises.

However, existing homology detection methods have some limitations. For instance, polymorphic malware can change its appearance and behavior to evade detection, and modified or obfuscated malware can be difficult to detect. Additionally, these methods may have difficulty identifying homologous malwares that share similar functionality but belong to different families. Currently, one widely used method is based on binary code similarity [5]. However, this approach suffers from a high rate of false positives and missing positives. False positives occur because functions with similar code structures and semantics may have different purposes and behaviors, such as functions that handle different types of inputs. On the other hand, underreporting is mainly caused by binary programs coming from different compilers and optimization levels, which can cause the same source code to behave differently in different binary programs.

To improve the accuracy of binary code similarity detection, various solutions have been proposed, including finer-grained approaches based on basic blocks [6–8] and deeper semantic approaches based on natural language processing [9–12]. However, none of these approaches fully leverage the information about the modular structure of the program. In fact, malwares of the same family often reuse the same functional modules, such as network communication, encryption, and Trojan horses, to improve development efficiency and quality. This provides a crucial basis for the homology determination of malicious codes. Therefore, we can identify similar functional modules in malwares to detect homology. Based on this idea, this paper proposes a coarse-grained module-based code similarity detection method, ModDiff. This method first uses graph embedding to classify a binary malicious code into function-based modules and then detects the homology of the malicious code by comparing the similarity between modules. Compared with function-level similarity detection, module similarity analysis has higher granularity and a more comprehensive view, which can effectively identify members of malware families and quickly categorize new malwares into known malware families when they are discovered.

The main contributions of the paper are summarized as follows:

- We propose a binary program modularity algorithm that decomposes programs into function-based modules.
- We propose a method for detecting function similarity using Siamese BERT networks.
- We propose a module similarity detection method for detecting the similarity of binary program modules.

1.1. Motivating Example

This section aims to highlight the crucial role of module similarity analysis in homology detection with a real-world example of two IoT botnets: Mirai [13] and Gafgyt [14], the basic information of which is presented in Table 1. These botnets are notorious for launching distributed denial of service (DDoS) attacks on a massive scale. Recent research reveals that several variants of Gafgyt reuse some of the Mirai modules, including HTTP flooding, UDP flooding, TCP flooding, STD, and the Telnet brute force module. It indicates that malware developers often adopt existing code modules while designing new attack tools to lower developmental costs and improve attack efficiency. This underlines the importance of module similarity analysis in detecting homologous malware.

Table 1. Information of samples.

Family	Mirai	Gafgyt
MD5	cd3b462b35d86fcc 26e4c1f50e421add	4b94d1855b55fb26 fc88c150217dc16a
Popular threat label (VirusTotal)	Trojan.linux/mirai	Trojan.linux/gafgyt
File size	160.84 KB	95.79 KB

Furthermore, analyzing malware modules can provide researchers with a better understanding of how the malware operates and attacks, which can help in targeting and improving preventive measures to enhance network security. For instance, if security experts identify that the code module of malware is similar to that of a previously known malware, they can anticipate how the malware will attack and take defensive measures to reduce the harm caused by the attack. Additionally, program module similarity analysis can enable security vendors to create more precise and effective malware detection and analysis tools that assist users in identifying and addressing potential security threats promptly, further enhancing network security. The technology of program module similarity analysis can be utilized by security vendors to classify and recognize malware, enhancing the ability to prevent and combat it.

1.2. Background Information

This section presents a brief background to the paper.

Binary Code Modularization. In software reverse engineering, binary code modularization techniques can assist analysts in obtaining a better understanding of the structure and function of a program. Generally, a program is composed of numerous code segments, each of which performs a particular function. Partitioning a binary program into multiple modules allows each code segment to correspond to a module, making the program easier to comprehend and analyze. Analysts can examine the code of each module to identify potential vulnerabilities or security issues in the program. This technique also allows developers to better organize and manage the code of their programs, improving the maintainability and reusability of the code.

The Siamese Framework. The Siamese network framework [15] is a deep learning-based framework that facilitates the comparison and matching of input data. In contrast to traditional single neural networks, the Siamese network framework comprises two identical neural network models with shared weights and parameters, resulting in greater efficiency and reduced overhead during training. In the Siamese network framework, the two neural network models process the two input data points separately, producing the corresponding feature vectors. These feature vectors can be compared for supervised learning, allowing comparison and matching of the input data. The strength of the Siamese network lies in its excellent generalization capability, enabling it to produce good results even when confronted with limited data sets.

The BERT Network. BERT [16] is a pre-trained language model that utilizes the transformer architecture and is widely employed for natural language processing tasks, such as text classification, named-entity recognition, and question-answering. By fine-tuning BERT, it can be tailored to specific downstream tasks. Fine-tuning entails adding new output layers or adapting existing output layers to a BERT model to suit the requirements of a specific task, and enhancing the model's accuracy in that task with a small amount of training. It enables BERT to be better adjusted to particular linguistic contexts and task specifications, thereby improving its efficiency in practical applications. In this paper, the BERT network is fine-tuned to capture the deep semantic information of assembly instructions and compute function similarity using Siamese networks.

2. Related Work

This section discusses work related to program modularization and code similarity detection.

2.1. Program Modularization

In the field of program analysis, software modularization refers to the division of a complex program into several relatively independent, cohesive, and loosely-coupled local parts via clustering basic units such as functions or classes [17]. These partial parts are named modules, which communicate and collaborate through interfaces between modules and eventually combine to complete the functionality of the entire software system. Depending on the object of the division, it can be divided into two ways: source code modularization and binary code modularization.

Source code modularization. Hong Xia et al. [18] proposed a hierarchical clustering combination method for clustering software modules. They used principal component analysis to combine the results of multiple hierarchical clusters, and the combined results retained as much basic information of each clustering algorithm as possible to achieve the best clustering results. Marios Papachristou [19] employed the Doc2Vec algorithm and call relations at the module level to construct a network graph and discover the community structure within it using the Louvain algorithm. Similarly, Weifeng Pan et al. [20] proposed a generalized Kernel based on a weighted directed graph to represent software topology, as traditional software module clustering using undirected unweighted graphs can result in a loss of information. They applied a decomposition method based on the weighted

directed graph to represent the software topology and ranked the classes according to the generalized core of the module to identify the key classes in the software.

Binary code modularization. A significant amount of program semantics is lost when the software source code is compiled into an executable file. Therefore, the first step in binary code module partitioning is to extract any remaining semantic information from the code and then apply a clustering algorithm to reasonably decompose the executable file into several components. BCD [21] is a method that uses static analysis techniques to decompose executables into modules. It takes functions as nodes and constructs decomposition graphs through three relationships: code location, data references, and function calls. Then, Newman's generalized community detection algorithm [22] is applied to divide the executable file into independent components. ModX [23] proposes a module quality metric based on specific program tuning, which can provide an accurate portrayal of the code structure, function calls, and other features in the program, thereby providing a reference for subsequent module division. The fast-unfolding Louvain algorithm [24] with heuristic bias is then used to decompose the binary file into modules.

2.2. Binary Code Similarity Detection

Binary code similarity involves comparing two or more binary codes to determine their similarities and dissimilarities. Several methods have been proposed for detecting binary code similarity.

Early research mainly tended to use bytecodes to compute similarity. IDA FLIRT [25] identified library functions via extracting bytecodes to generate fingerprints for library functions. DiscovRe [26] represented basic blocks via extracting statistical features and computed similarity based on the maximum common subgraph isomorphism problem. Bindiff [27] and Genius [28] extracted code features from control flow graphs and measured the similarity of binary functions based on graph isomorphism. BinGo [29] and IMFSim [30] captured the behavior of binary functions by sampling them with random values.

The graph-embedding approach enables the projection of a graph into a low-dimensional vector space while preserving the intrinsic structural properties of the graph, offering a less complex method for processing graph data. Gemini [31] extracts the attribute control flow graph of the function and trains the graph-embedding network to generate the embedding. VulSeeker [32] first constructs the semantic flow graph of the token and extracts the basic block features as the two vectors of functions. Finally, it generates the embedding vectors of the entire binary functions via the use of semantic-aware DNN models.

Recent research in binary code similarity detection has been influenced by natural language processing. Asm2vec [9] learns semantic information between tokens using the PV-DM model [33] and represents assembly functions using a weighted mixture of semantic information. Safe [10] employs skip gram to generate instruction embeddings to overcome the problem of missing important features due to manual feature extraction. PalmTree [11] is the first approach with which to apply BERT to instruction embedding, demonstrating the potential of language models in BCSD. To fully extract the structural information of a program, jTrans [12] embeds control flow information of binary code into a transformer-based language model and applies it to code similarity detection.

3. Methodology

3.1. Overview

The workflow of ModDiff is displayed in Figure 1. The process of detecting similarity between malware modules involves two main components: program module partitioning and module similarity calculation. Initially, two binary programs are transformed into attribute graphs and divided into function-based modules using a module partitioning algorithm based on graph embedding. Subsequently, the modules requiring similarity computation are selected and the similarity of all functions between them is determined using a Siamese BERT network. The next step involves constructing a bipartite graph with the functions in the modules as vertices and the inter-function similarity as weights. Then,

a matching algorithm is applied to find the maximum weight match of the bipartite graph, which yields the similarity of the modules. Eventually, the homology decision based on the module similarity threshold is used to determine whether or not two malicious codes belong to the same family.

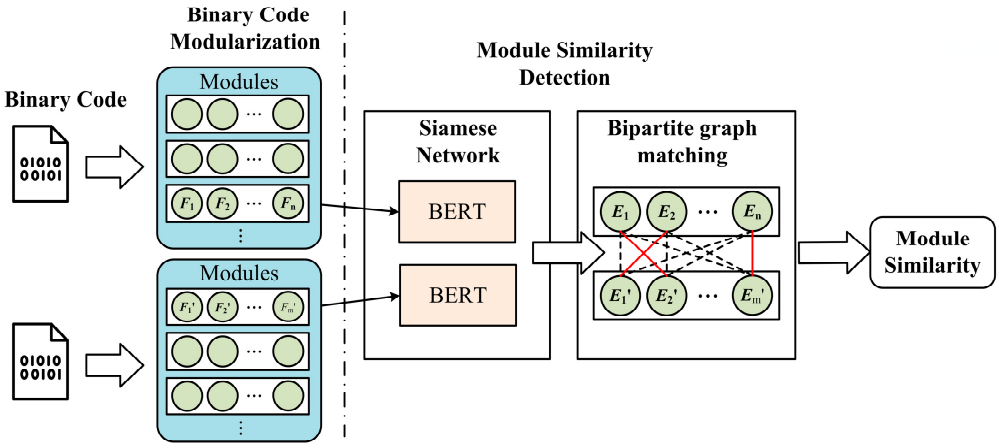

Figure 1. Schematics of of ModDiff.

3.2. Binary Code Modularization

Modularity is a crucial aspect of software design, wherein functions that perform similar tasks are grouped into a module. To leverage the modularity information in binary programs to its fullest potential, this paper proposes a deep learning-based approach that employs a graph-embedding clustering algorithm to cluster functions and achieve software module classification. Concretely, we first fully parse the software using hybrid disassembly and construct a function attribute graph. The function nodes in the graph are then encoded using a graph self-encoder to obtain their embedding representation. Finally, a goal-directed clustering algorithm is applied to cluster the nodes and achieve software module classification.

3.2.1. Attribute Graph Construction

In order to enhance the precision of module division, our approach involved analyzing the features that express modular structure information in programs, taking into account software design and compilation principles. After careful consideration, we selected four features that were deemed most relevant for this purpose. **Function addresses**: During the process of converting the source code into executable files, compilation optimization algorithms may significantly impact the program structure. However, it has been observed that the original sequential relationships between function locations are largely preserved [34]. As a result, functions with similar addresses are more likely to belong to the same module. **Function calls**: According to the single responsibility principle, each module should ideally serve a single function. Therefore, a group of functions that frequently call each other are likely to belong to the same module. **Data references**: Well-designed modules are based on the principle of high communication cohesion and low public coupling. As such, they try to avoid using public data between them. In other words, functions that access the same data area are highly likely to belong to the same module. **API calls**: Software is typically divided into various modules based on their intended functionality. Functions within a module are expected to use a sequence of semantically similar APIs to accomplish their respective tasks. As a result, functions that use semantically similar APIs are more likely to belong to the same module cluster. Following the pre-processing of the software to obtain function and feature data, we construct a directed attribute graph with functions serving as nodes, call relationships between functions as directed edges, and four types of feature information as node attribute data.

3.2.2. Attribute Graph Embedding

The graph-embedding algorithm was designed to map high-dimensional graph data into low-dimensional vectors while maximizing the retention of valid information from the original data. This enables the efficient analysis and application of the data in subsequent steps. To capture the relationships between nodes more comprehensively, ModDiff employs a graph attention self-encoder [35] to embed the nodes into a low-dimensional space, as shown in Figure 2.

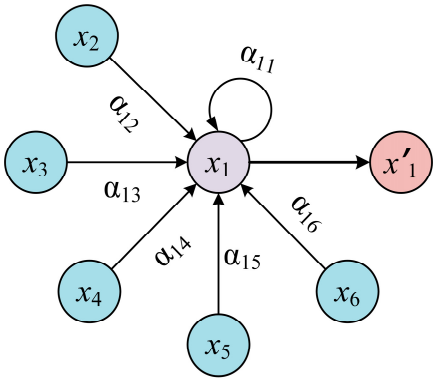

Figure 2. Structure of the graph attention self-encoder.

The primary steps of the graph attention self-encoder are as follows: Initially, it takes the attribute information of a node as the initial representation of the node. Then, it utilizes the graph attention mechanism to calculate the importance of neighboring nodes to the current node, known as the attention coefficient. Lastly, the final node representation is obtained by multiplying the embedding representations of all neighboring nodes by the attention factor and adding the embedding representation of the current node.

Specifically, given the attribute graph $G = (V,E,X)$, where V denotes the set of nodes, E denotes the set of edges and X denotes the set of node attributes, each node $v_i \in V$ has a d-dimensional attribute vector $x_i \in \mathbb{R}^d$ for characterizing the nodes. The goal of the graph attention self-encoder is to map the attribute information and structural information of each node, v_i, to a k-dimensional vector, $\boldsymbol{x}_i' \in \mathbb{R}^k$, for the next step of analysis and application.

To achieve this goal, the graph attention self-encoder employs an attention mechanism that computes the importance of each node's neighboring nodes to the current node. Given a node, v_i, and its set of neighboring nodes, N_i, the attention mechanism calculates the importance of each neighboring node, $v_j \in N_i$, to the node, v_i; namely, the attention coefficient, α_{ij}:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\vec{a}^{\mathrm{T}}[\mathbf{W}\boldsymbol{x}_i \| \mathbf{W}\boldsymbol{x}_j]))}{\sum_{k \in N_i} \exp(\text{LeakyReLU}(\vec{a}^{\mathrm{T}}[\mathbf{W}\boldsymbol{x}_i \| \mathbf{W}\boldsymbol{x}_j]))}. \tag{1}$$

where $\mathbf{W} \in \mathbb{R}^{F' \times F}$ represents the shared weight matrix, and $\vec{a} \in \mathbb{R}^{2F'}$ is a weight vector, the role of which is to map the stitched high-dimensional features to a real number. Finally, the embedding representations of all neighboring nodes are multiplied by the attention factor and added to the embedding representation of the current node to obtain the final node representation:

$$\boldsymbol{x}_i' = \sigma(\textstyle\sum_{j \in N_i} \alpha_{ij} \mathbf{W}\boldsymbol{x}_j), \tag{2}$$

where $\boldsymbol{x}_i'$ denotes the output representation of node, v_i, and σ denotes a nonlinear function. After encoding the graph, ModDiff optimizes the embedding representation of the nodes with a self-training module based on reconstruction losses to obtain a more accurate representation.

3.2.3. Target-Oriented Node Clustering

To improve the accuracy and efficiency of clustering, ModDiff adopts a target-oriented clustering algorithm [36]. The core idea of the algorithm is to guide the clustering process by introducing an auxiliary target distribution to obtain more accurate and efficient clustering results. Specifically, the algorithm initially operates the k-means algorithm multiple times based on the node embedding representation to obtain more accurate initial clustering centers. Then, Student's T-distribution is employed as a kernel function to measure the similarity between each data point and the clustering center, which is applied as a soft clustering label for each category to which the data point belongs:

$$q_{iu} = \frac{\left(1 + \|x_i - \mu_u\|^2/\alpha\right)^{-\frac{\alpha+1}{2}}}{\sum_k \left(1 + \|x_i - \mu_k\|^2/\alpha\right)^{-\frac{\alpha+1}{2}}}, \tag{3}$$

where $\boldsymbol{x}_i$ denotes the embedding of node, v_i, μ_u is the cluster center of u, α is the degree of freedom of the T-distribution, and q_{iu} represents the probability that the node, v_i, belongs to the cluster u.

Considering that soft clustering labels with high probability have high confidence, the algorithm raises the actual distribution of nodes to a quadratic one and normalizes it as the target distribution:

$$p_{iu} = \frac{q_{iu}^2/\sum_i q_{iu}}{\sum_k \left(q_{ik}^2/\sum_i q_{ik}\right)}, \tag{4}$$

where p_{iu} denotes the target probability that the node, v_i, belongs to u. Finally, the algorithm calculates the KL divergence between the actual distribution of data points and the target distribution as the clustering loss. The model parameters are updated to minimize the KL divergence loss using the Adam optimization algorithm [37] until the maximum number of iterations is reached. The final label of each node is obtained via the optimized Q distribution.

3.3. Module Similarity Detection

Following the modularization of the program, we propose a module similarity detection algorithm based on function similarity. The algorithm comprises two stages. Firstly, it competes the similarity between functions via Siamese networks, and secondly, calculates the similarity between modules using the graph matching algorithm.

3.3.1. Function Encoding via Siamese Network

To accurately evaluate the similarity of functions and capture deeper semantic information about functions, ModDiff leverages the Siamese BERT networks [38] to compute the similarity of functions. Figure 3 presents the network architecture of the Siamese BERT networks. Given two binary functions, ModDiff first preprocesses them via normalizing instructions, tokenizing assembly, and serializing functions [39]. The processed functions are then fed into two BERT encoders with identical structures and parameters. The output from the encoder is converted into a fixed-size vector through a mean pooling operation. Finally, the similarity between the two functions is computed by measuring the cosine similarity of the output vectors.

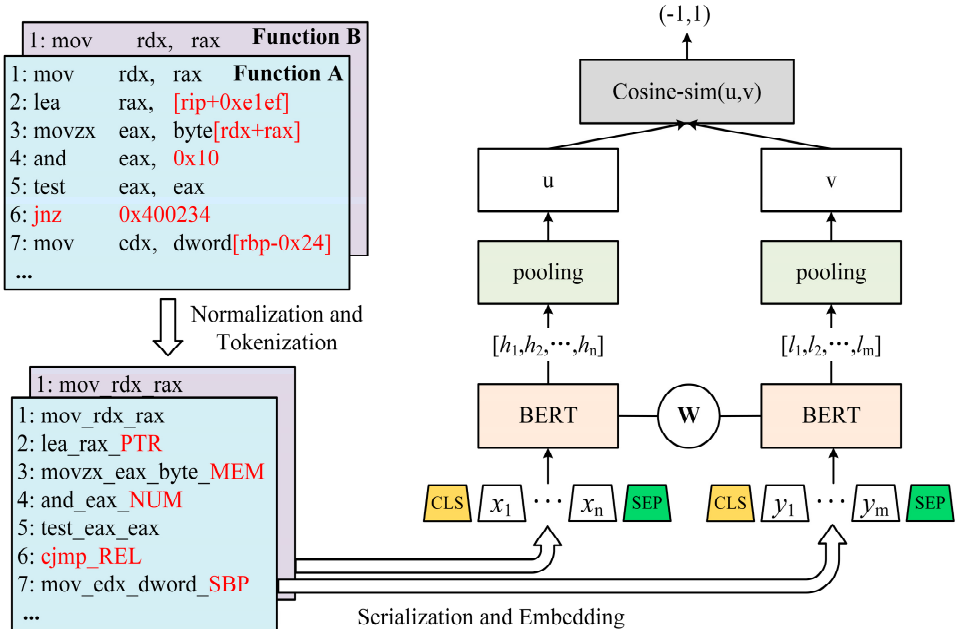

Figure 3. Siamese BERT network architecture, where h_i and l_j denote the hidden representation of the i-th and j-th assembly instructions in the input function, respectively.

Preprocessing

As the raw instructions of assembly functions contain a large amount of specific machine code and syntax structures, these are difficult to understand by natural language models. Therefore, ModDiff preprocesses the input functions to make them more compatible with natural language model processing.

The first step in the preprocessing of assembly functions is instruction normalization, which aims to convert information such as constants, addresses, and jumps in instructions into a standardized expression. The principles for instruction normalization are outlined in Table 2.

Table 2. Principles of instructional normalization.

Original Instructions	Normalized Expression
Indirect addressing with register "eip/rip"	PTR
Indirect addressing with register "esp/rsp"	SSP
Indirect addressing with register "ebp/rbp"	SBP
Other indirect addressing	MEM
Relevant addressing	REL
Immediate number	NUM
Float instruction with register "xmm"	XMM
Conditional jump	cjmp

After instruction normalization, the previously complex and variable instructions in assembly functions are converted into a relatively uniform expression. The following step is instruction tokenization, where the input instructions are partitioned into meaningful token units according to specific rules. This enables the model to process and analyze the input information more effectively. To comprehensively and accurately capture the semantic information of each instruction, we employ a full-instruction-level tokenization approach, treating each assembly instruction as a separate and fully meaningful unit.

The last step in preprocessing is function serialization, which transforms the structured function into a sequence of tokens that can be further processed and analyzed by the model. As the semantic information in the assembly instructions already includes the structured information of the program, ModDiff serializes the instructions directly in

address order. In particular, each instruction is traversed within the function in ascending address order and tokenized at the full instruction level, resulting in a serialized sequence of instruction tokens.

Backbone network

Siamese BER networks is a model built on the Siamese network framework, comprising two BERT networks [16] that share weights. Our purpose is to fine-tune the BERT networks in the Siamese network framework to generate an assembly function embedding vector that is more suitable for the task of function similarity computation. To achieve fixed-size function embeddings, we apply an averaging pooling operation to the output of the BERT networks. This approach could convert assembly functions of different lengths into a fixed-length vector representation, facilitating subsequent similarity calculations.

Loss function

The loss function plays a crucial role in fine-tuning the model, which determines the performance of the model in a particular downstream task. For each pair of assembly functions, we first generate the embedding vectors u and v via Siamese BERT networks, then calculate the similarity of the two vectors using cosine similarity and compare the results with the true similarity labels. The model is fine-tuned via back-propagation losses to make it more suitable for the task of compilation function similarity calculation. As the mean square error has the advantages of being easy to compute, highly interpretable and robust against outliers, it is chosen as the loss function for measuring the difference between the true and predicted labels:

$$loss_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2, \tag{5}$$

where x_i is the true label and y_i represents the cosine similarity of the two function-embedding vectors obtained through the model.

Datasets

During the training process, we utilized a training set comprising widely used and representative projects including Coreutils-9.0, Curl-7.82, Diffutils-3.8, Findutils-4.80, Binutils-2.37, Tcpdump-4.93, and Gmp-6.2.1. In order to create a comprehensive dataset, we compiled the aforementioned projects using two compilers (GCC-10 and Clang-10) and four optimization levels (O0, O1, O2, and O3). This resulted in close to 500 K function pairs, with 90% being allocated to training and the remaining 10% being allocated to verification. The use of multiple compilation options enabled us to obtain more precise and extensive training data, which ultimately led to the improved performance of the machine learning models.

3.3.2. Module Similarity Calculation

This section presents a novel approach to module similarity measurement based on function similarity. To reduce ambiguity, we first define the module similarity problem formally.

Definition 1. *The module similarity problem. Given two program modules,* $X = \{x_i\}_{i=1}^{n}$ *and* $Y = \{y_i\}_{i=1}^{m}$, *with* $n < m$, *any function in X and any function in Y satisfies* $0 \leq FuncSim(x, y) \leq 1$. *The module similarity problem can be defined as finding a single projection,* $f : X \to Y$, *from set X to set Y such that* $\sum_{x \in X} FuncSim(x, f(x))$ *achieves a maximum value*.

In order to measure module similarity more accurately, we further propose a definition of module similarity.

Definition 2. *Module similarity. Due to the varying number of functions between modules, we define* $[\sum_{x \in X} FuncSim(x, f(x))]_{\max}/|X|$ *as the similarity of module X to module Y, and* $[\sum_{x \in X} FuncSim(x, f(x))]_{\max}/|Y|$ *as the similarity of module Y to module X*.

To solve the module similarity problem, this paper employs the concept of the bipartite graph-matching problem and utilizes its matching algorithm to perform the necessary computations.

Typically, ModDiff models the modules to be detected in both programs as an entitled bipartite graph, $G = (X, Y, E)$, where X and Y represent the program modules to be detected (a set of clusters of functions and the nodes in the set are not adjacent).

The set of edges, E, is constructed as follows: for $\forall x \in X, y \in Y$, if $FuncSim(x, y) > 0$, then an undirected edge $\langle x, y \rangle$ of weight, $FuncSim(x, y)$, is added between the two vertices corresponding to x and y in the bipartite graph, G. After modeling by a bipartite graph, the module similarity problem is equivalent to solving the maximum weight-matching problem on the weighted bipartite graph.

ModDiff utilizes the Kuhn–Munkres (KM) algorithm [40] to find the maximum weight match. The KM algorithm is a classical and efficient algorithm for solving the best match of entitled bipartite graphs via solving the perfect match of equivalent subgraphs of the bipartite graph. However, it is only applicable when the number of nodes in X and Y is equal, which is not always the case in our problem. To overcome this limitation, we propose a module similarity metric algorithm based on the KM algorithm that can handle different numbers of nodes. The main flow of the algorithm is shown in Algorithm 1. This algorithm can effectively calculate the similarity of modules quickly and accurately, even when the number of nodes is unequal.

Algorithm 1 Module similarity calculation algorithm

Input: $G = (X, Y, E)$ and $|X| \leq |Y|$
Output: Module Similarity
if $|X| < |Y|$ **then**
  Add $(|Y| - |X|)$ virtual nodes to X
  for virtual node i in X **do**
    Add virtual edges between virtual node i and all node in Y
    Set the weight of new virtual edges to zero
  end for
end if
Use KM algorithm to find an maximum weight matching M and maximum weight W for G
Take $W/|X|$ as the similarity of module X to module Y
Take $W/|Y|$ as the similarity of module Y to module X

After acquiring the module similarity of the malware, we can establish the module similarity threshold with expert experience or experiments. When the module similarity of two malwares surpasses this threshold, they are deemed to be part of the same family.

4. Evaluation

The experimental section aims to answer the following research questions:
RQ1: How accurate is the binary code modularity approach in ModDiff?
RQ2: How accurate is ModDiff in malware homology determination?
RQ3: How robust is ModDiff?
RQ4: How scalable is ModDiff?

To evaluate the performance of ModDiff, we compare it with two widely used binary modularization tools and examined its accuracy in detecting malware homology. However, given the diversity of malwares, we also evaluate the robustness of ModDiff under different compilers and optimization options. In addition, we evaluate the scalability of ModDiff by comparing the similarity between multiple versions of software.

4.1. Binary Code Modularization Evaluation (RQ1)

Datasets. To construct the datasets for evaluating the accuracy and scalability of the binary code module partitioning method in ModDiff, we compiled a source code from open-source software repositories and official software websites to obtain executable programs, and constructed three datasets, as illustrated in Table 3. S1 consists of open-source software

developed and maintained by organizations or volunteers working together, obtained from official software websites. S2 is a collection of open-source software designed and developed by individuals or small teams, obtained from the GitHub open-source repository. M1 is a dataset consisting of 728 common pieces of malware that were extracted using the SourceFinder [41] method.

Table 3. Datasets for modularization evaluation.

Dataset	Samples	Average Functions	Average Clusters	Average Size (KB)
S1	82	847	17	979
S2	1056	931	16	883
M1	728	256	13	325

To conduct a comprehensive evaluation of the performance of ModDiff in binary code module partitioning, we compared it with two mainstream binary code modularization methods, BCD [21] and ModX [23], for comparison and conducted experiments on three datasets, respectively. As the source code for two algorithms was not readily available, we implemented a simple prototype version based on the descriptions provided in the relevant literature.

Table 4 presents the experimental results of the use of these methods on the three datasets. ModDiff outperforms the other methods in terms of Prc, NMI, and F1 metrics for all three datasets. Specifically, ModDiff achieves an average F1 score of 0.802, 0.701 and 0.692 on datasets S1, S2, and M1, respectively, which are significantly better than the scores achieved via BCD and ModX. This suggests that the graph embedding-based module partitioning approach can better extract semantic information from functions and recover the modular structure of the program. Comparing the results for S1 and S2 reveals that well-designed software can be more easily reverse-decomposed into modules. Furthermore, we found that ModDiff performs exceptionally well on the M1 dataset, indicating its effectiveness at dealing with unencrypted and obfuscated malwares.

Table 4. The modularization results.

Dataset	Method	Prc	NMI	F1
S1	BCD	0.623	0.505	0.641
	ModX	0.648	0.547	0.687
	ModDiff	**0.783**	**0.737**	**0.802**
S2	BCD	0.539	0.498	0.589
	ModX	0.592	0.512	0.635
	ModDiff	**0.703**	**0.687**	**0.701**
M1	BCD	0.539	0.513	0.557
	ModX	0.571	0.532	0.633
	ModDiff	**0.632**	**0.674**	**0.692**

Answer to RQ1: compared to other dominant binary modularization tools, ModDiff has better precision and a better F1 score, and can restore the modular structure of a program more accurately.

4.2. Homologation Detection Accuracy Evaluation (RQ2)

In this section, we assess the performance of the ModDiff method in homology detection via malware variant detection.

Datasets. To comprehensively evaluate the effectiveness of ModDiff, we constructed a dataset comprising 186 families of active malware variants obtained from Malware Bazaar [42]. The samples in this dataset met the following conditions: (1) absence of code protection measures such as obfuscation and shelling, (2) utilization of the X86 architecture, and (3) presentation of at least three samples in each family. With these filtering criteria, we generated a dataset that comprised 897 malicious binaries.

Ground Truth. For the purpose of evaluating the module similarity results, we defined two samples to be considered similar, which means they belong to the same malicious family, if the similar modules (module similarity above T_{ModSim}) exceed T_{ModPer} in both samples, where T_{ModSim} is the module similarity threshold and T_{ModPer} is the module similarity number percentage threshold.

In the experiments, we first applied a binary code module partitioning algorithm to decompose all samples into function-based modules, and then used the module similarity algorithm to calculate the similarity between any two modules. Based on the results of the similarity comparison, predicted labels for each module were derived and compared with the true labels to assess the accuracy of the algorithm.

Figure 4 shows the performance of ModDiff for the malware variant detection task at different T_{ModSim} and T_{ModPer} values. Our results show that when T_{ModSim} is 0.8 and T_{ModPer} is 0.7, ModDiff achieves an F1 score of 0.89 for classification, which is highly accurate in identifying malware of the same family. Nevertheless, further analysis revealed that excessively high or low threshold values can impair the performance of the model. When the threshold is set too low, the model may have a higher false positive rate, which means that samples that do not belong to the same family may be incorrectly classified into the same family. This can lead to inaccuracies in classification results for different families. Conversely, if the threshold is set too high, the model may have a high false negative rate, which means that samples belonging to the same family may be classified into different families. This can result in inaccuracies in classification results for the same family. Therefore, when training and applying the model, it is important to take into account the trade-off between the false positive and false negative rates based on the specific circumstances, and select suitable threshold parameters to achieve optimal classification results.

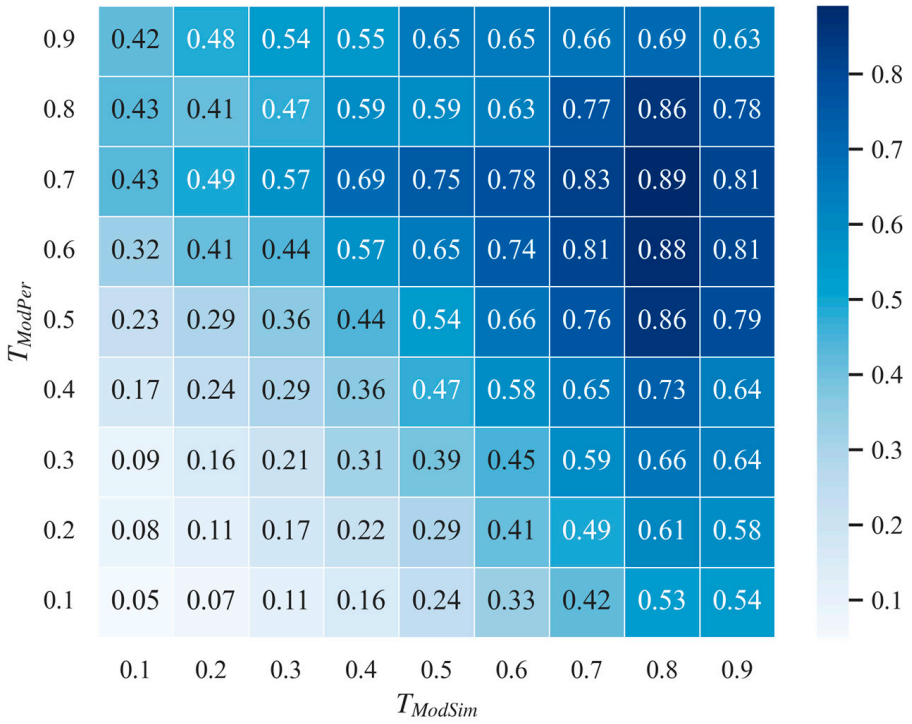

Figure 4. F1 scores with different T_{ModSim} and T_{ModPer} values.

Answer to RQ2: With reasonable threshold settings, ModDiff achieves an F1 score of 0.89 in malware family classification experiments, indicating that the method can accurately determine the homology of malware.

4.3. Robustness Evaluation (RQ3)

The robustness of ModDiff mainly encompasses compiler and optimization-level robustness. To evaluate robustness, we recompiled the malicious sample dataset M1 collected in Experiment 1 (Section 4.1) using two compilers (GCC-10 and Clang-10) and four optimization levels (O0, O1, O2, and O3). This resulted in a test set of 5824 binaries.

The robustness evaluation experiment consisted of two parts. The first part involved comparing binaries generated by two compilers at the same optimization level to determine whether or not they were compiled and generated from the same source code. The second part involved comparing binaries generated by O1, O2, and O3 with the O0 level, respectively, at the same compiler to determine whether or not they were the same malware.

Figure 5 presents the results of the ModDiff comparison of binaries generated by different compilers at four optimization levels. Overall, the F1 score of the model tends to decrease as the compilation optimization level increases. This indicates that the optimization algorithms of different compilers affect the modular structure and functions of the program differently, making it difficult for the model to recognize the same binaries. However, further analysis reveals that as the optimization level increases, the precision of the model also gradually increases, indicating that the model is able to recognize the same binary files more accurately. On the other hand, the decrease in recall suggests that the model may miss some identical binaries at high optimization levels. Therefore, in practical applications, a balance must be struck and adjustments must be made to ensure that the model performs optimally.

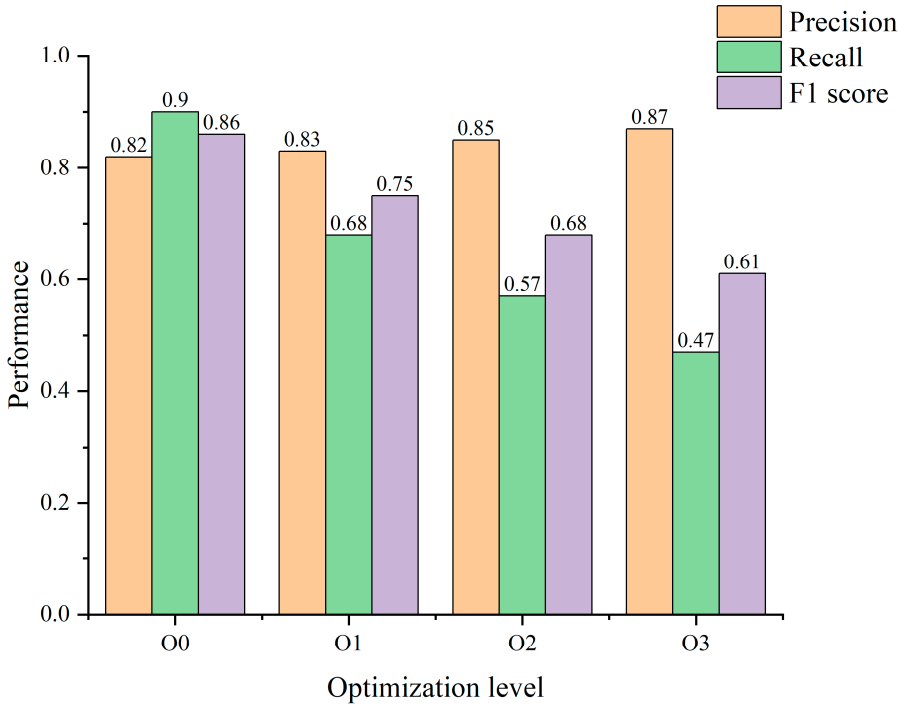

Figure 5. Performance on different compilers.

The performance of ModDiff at different compilation optimization levels is shown in Table 5. Intuitively, we observe that when the optimization level is increased from O1 to O3, the detection accuracy of the model in GCC and Clang-generated code decreases by 0.18 and 0.21, respectively. The results show that as the compilation optimization level increases, the precision of the model also gradually increases, indicating that the model is able to recognize the same binary files more accurately. On the other hand, the decrease in recall suggests that the model may miss some identical binaries at high optimization levels. This trade-off between precision and recall highlights the importance of selecting appropriate threshold values to achieve accurate and reliable classification results.

Table 5. F1 score on different optimization levels.

	O0 vs. O1	O0 vs. O2	O0 vs. O3	Avg
GCC	0.83	0.76	0.65	0.75
Clang	0.81	0.72	0.60	0.71

Answer to RQ3: The experimental results show that although the F1 value of the model shows a decreasing trend in experiments with different compilers and optimization levels, the average accuracy of the model reaches 0.73. This indicates that ModDiff is able to perform effective binary file comparison in different compilation environments with some generality and adaptability.

4.4. Scalability Evaluation (RQ4)

To confirm the suitability of ModDiff for detecting homologation in common software, we compiled various versions of the source code using the same compiler and optimization options. This ensured that any differences between the compiled binaries were solely due to version changes. Then, these binaries were examined using the ModDiff algorithm to determine whether or not they belong to different versions of the same software. Furthermore, we compared ModDiff with two commonly used binary comparison tools, BinDiff [43] and Diaphora [44], to comprehensively evaluate its effectiveness.

Table 6 displays the open-source projects commonly used under Linux that we collected as datasets. These projects are diverse in their application types and are widely used. We downloaded multiple versions of each project from their official websites and selected two versions as our dataset. To evaluate the scalability of ModDiff, we specifically chose two versions that were released over five years apart from each other, ensuring that the differences between them were significant.

Table 6. Datasets for scalability evaluation.

Datasets	Version		Binary Pairs
Binutils	2.30	2.40	16
Coreutils	6.6	9.0	97
Diffutils	2.9	3.8	4
Findutils	4.2	4.9	4

The results of the scalability evaluation are presented in Table 7, indicating that ModDiff performs exceptionally well on the open-source projects with an accuracy rate of 93%. This accuracy rate is higher than that of other binary comparison tools, demonstrating the superior performance of ModDiff in general software homologation detection. In addition, we also analyzed the false positive and false negative rates of ModDiff, which were found to be very low. This indicates that ModDiff is highly accurate in detecting homologous software in open-source projects, while minimizing the risk of misclassification. Furthermore, our method also consumes less time and computational resources during the detection process, allowing for quick completion of the task.

Table 7. Performance on scalability evaluation.

	Precision	Recall	F1
ModDiff	0.94	0.92	0.93
BinDiff	0.90	0.80	0.85
Diaphora	0.91	0.85	0.88

Answer to RQ4: ModDiff outperforms other mainstream binary matching tools on a dataset built with common software and accurately identifies different versions of binary files, exhibiting excellent scalability.

5. Discussion

In this section, we first illustrate other potential applications of the modular similarity analysis technique, and then present the limitations of the approach as well as future work.

5.1. Applications

ModDiff enables modular partitioning of a binary code even without the source code or debugging information, and can identify similar modules across diverse executables. In addition to its crucial role in malware homologation detection, ModDiff has other potential applications.

Code plagiarism detection. With the rapidly increasing number of open-source projects, code plagiarism has become a serious issue in the software industry, posing a significant threat to its sustainable growth [45]. Code plagiarism refers to the unauthorized use of the software code of others for commercial purposes or other unapproved uses without obtaining a license. The module similarity analysis method of ModDiff provides a new detection granularity for code plagiarism detection. By dividing the software into function-based modules and identifying similar modules using a module similarity algorithm, ModDiff can detect code plagiarism. Compared to traditional detection techniques based on function or basic block similarity, the module-based approach can recognize similar functional modules in an application, resulting in higher detection accuracy and lower false alarm rates.

Software vulnerability detection. Software **vulnerability** detection aims to prevent or mitigate security vulnerabilities that may be exploited by attackers, ultimately safeguarding information assets. Currently, many function similarity-based methods are proposed for software security detection [46]. However, module similarity-based detection methods are more effective as they can determine the similarity between code modules by comparing their structural, semantic and execution path characteristics, thus improving the accuracy of detection. Therefore, the module similarity-based detection method plays a crucial role in software vulnerability detection.

Accelerated reverse analysis. As software functionality and complexity increase, the size of executables also increases, posing significant challenges to the speed and accuracy of reverse analysis. Binary-code module division techniques can help analysts divide software into relatively independent functional modules, enabling them to recover the modular control structure of the program. This facilitates the swift identification of code locations for specific functions, thereby expediting the process of reverse analysis. In addition, in the field of malware analysis, module partitioning can help analysts extract modules such as encryption, packaging, and communication in malware [47]. The speed of automated malware analysis can be increased via implementing module reuse in a secure and controlled environment to extract an execution code, analyze communication protocols and capture traffic characteristics.

5.2. Limitation and Future Work

Despite its potential, the ModDiff approach has some limitations that need to be addressed in future research.

Firstly, code obfuscation can significantly alter the control structure and data dependency structure of a program, presenting a significant challenge in the semantic extraction of functions. This challenge poses two problems for the ModDiff approach. The first issue pertains to the impact on the modularization of the code, as the function call and data reference features extracted via the ModDiff method are vulnerable to code obfuscation, which affects the accuracy of the modularity. The second relates to functional similarity calculation, as the Siamese BERT network-based function similarity calculation method used in this paper has a high miss rate when dealing with code obfuscation. In the future, effective deobfuscation methods [48] need to be considered to recover the original code structure as much as possible.

Secondly, the module partitioning method used in this paper generates outcomes with uncertainty, which may affect the accuracy of the module similarity analysis. If the module partitioning is not precise enough, unrelated modules may be categorized into the same module, resulting in high similarity between multiple modules. Conversely, related modules may be classified into different modules, resulting in low-similarity analysis results. In the future, it may be beneficial to consider using expert knowledge to guide the partitioning process and ensure that the delineated modules are closer to the actual modules, thus improving the accuracy of module similarity analysis.

Thirdly, the accuracy of ModDiff primarily relies on the algorithm used for function similarity detection. The limitations of semantic learning models in function similarity computation arise mainly from two aspects. On the one hand, they may not accurately capture the relationship and semantic information between functions, resulting in inaccurate computed similarity. On the other hand, the models may be over-fitted or under-fitted due to a lack of sufficient semantic information or insufficient training data, resulting in distorted similarity calculations. In future research, it may be beneficial to utilize several different semantic learning models for comparison and to ensure that the model is adequately trained and validated to accurately capture the relationship between functions and semantic information. Additionally, other methods or features can be used to calculate function similarity to enhance the accuracy of the calculation results.

6. Conclusions

This paper proposes a method called ModDiff to detect the homology of malware based on module similarity. The method first decomposes programs into function-based modules using graph-embedding clustering. It then calculates the similarity of modules via matching the similarity functions in the modules. The experimental results demonstrate that the modularization technique of ModDiff can effectively partition programs into relatively independent modules, thereby improving the speed and accuracy of reverse analysis. Moreover, ModDiff exhibits excellent accuracy in the malware family detection task and good robustness and scalability in multiple sets of evaluation experiments, which confirms the practicality and reliability of the method.

Author Contributions: Conceptualization, H.S. (Huaqi Sun); methodology, H.S. (Huaqi Sun) and H.S. (Hui Shu); software, H.S. (Huaqi Sun) and Y.G.; validation, H.S. (Huaqi Sun) and F.K.; formal analysis, H.S. (Huaqi Sun) and H.S. (Hui Shu); investigation, H.S. (Huaqi Sun) and H.S. (Hui Shu); resources, H.S. (Huaqi Sun) and Y.G.; data curation, H.S. (Huaqi Sun) and F.K.; writing—original draft preparation, H.S. (Huaqi Sun); writing—review and editing, H.S. (Hui Shu); visualization, Y.G.; supervision, F.K.; project administration, H.S. (Huaqi Sun). All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The malware source code used in this paper was mainly obtained from open-source repositories, including https://github.com/ytisf/theZoo (accessed on 5 April 2023), https://virusshare.com (accessed on 5 April 2023), and https://bazaar.abuse.ch (accessed on 11 May 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. About Malware and Pua. Available online: https://portal.av-atlas.org/malware (accessed on 5 April 2023).
2. Almomani, I.M.; Ahmed, M.; El-shafai, W. Android malware analysis in a nutshell. *PLoS ONE* **2022**, *17*, e0270647. [CrossRef] [PubMed]
3. El-shafai, W.; Almomani, I.M.; Alkhayer, A. Visualized Malware Multi-Classification Framework Using Fine-Tuned CNN-Based Transfer Learning Models. *Appl. Sci.* **2021**, *11*, 6446. [CrossRef]
4. Almomani, I.; Alkhayer, A.; El-Shafai, W. An Automated Vision-Based Deep Learning Model for Efficient Detection of Android Malware Attacks. *IEEE Access* **2022**, *10*, 2700–2720. [CrossRef]
5. Haq, I.U.; Caballero, J. A Survey of Binary Code Similarity. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
6. Duan, Y.; Li, X.; Wang, J.; Yin, H. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020.

7. Xue, Y.; Xu, Z.; Chandramohan, M.; Liu, Y. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Trans. Softw. Eng.* **2019**, *45*, 1125–1149. [CrossRef]
8. Xu, Y.; Xu, Z.; Chen, B.; Song, F.; Liu, Y.; Liu, T. Patch based vulnerability matching for binary programs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing Analysis, Virtual Event, USA, 18–22 July 2020.
9. Ding, S.H.H.; Fung, B.C.M.; Charland, P. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 472–489. [CrossRef]
10. Massarelli, L.; Luna, G.A.D.; Petroni, F.; Querzoni, L.; Baldoni, R. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Saclay, France, 28–29 June 2018.
11. Li, X.; Yu, Q.; Yin, H. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In Proceedings of the ACM SIGSAC Conference on Computer Communications Security, Virtual Event, 15–19 November 2021.
12. Wang, H.; Qu, W.; Katz, G.; Zhu, W.; Gao, Z.; Qiu, H.; Zhuge, J.; Zhang, C. jTrans: Jump-aware transformer for binary code similarity detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing Analysis, Seoul, Republic of Korea, 18–22 July 2022.
13. Antonakakis, M.; April, T.; Bailey, M.; Bernhard, M.; Bursztein, E.; Cochran, J.; Durumeric, Z.; Halderman, J.A.; Invernizzi, L.; Kallitsis, M.; et al. Understanding the Mirai Botnet. In Proceedings of the USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017.
14. GAFGYT. Available online: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/GAFGYT/ (accessed on 5 April 2023).
15. Bromley, J.; Bentz, J.W.; Bottou, L.; Guyon, I.M.; LeCun, Y.; Moore, C.; Säckinger, E.; Shah, R. Signature Verification Using a "Siamese" Time Delay Neural Network. *Int. J. Pattern Recognit. Artif. Intell.* **1993**, *7*, 669–688. [CrossRef]
16. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2019**, arXiv:1810.04805.
17. Sarhan, Q.I.; Ahmed, B.S.; Bures, M.; Zamli, K.Z. Software Module Clustering: An In-Depth Literature Analysis. *IEEE Trans. Softw. Eng.* **2022**, *48*, 1905–1928. [CrossRef]
18. Xia, H.; Zhang, Y.; Chen, Y.; Zhang, H.; Wang, Z.; Wang, F. Software Module Clustering Using the Hierarchical Clustering Combination Method. In Proceedings of the 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), Chengdu, China, 22–24 April 2022; pp. 155–160. [CrossRef]
19. Papachristou, M. Software clusterings with vector semantics and the call graph. In Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019.
20. Pan, W.; Song, B.; Li, K.; Zhang, K. Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Gener. Comput. Syst.* **2018**, *81*, 188–202. [CrossRef]
21. Karande, V.; Chandra, S.; Lin, Z.; Caballero, J.; Khan, L.; Hamlen, K. BCD: Decomposing Binary Code into Components Using Graph-Based Clustering. In Proceedings of the Asia Conference on Computer and Communications Security, Incheon, Republic of Korea, 29 May 2018; pp. 393–398. [CrossRef]
22. Newman, M.E. Fast algorithm for detecting community structure in networks. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.* **2004**, *69 Pt 2*, 066133. [CrossRef]
23. Yang, C.; Xu, Z.; Chen, H.; Liu, Y.; Gong, X.; Liu, B. ModX: Binary Level Partially Imported Third-Party Library Detection via Program Modularization and Semantic Matching. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 25–27 May 2022; pp. 1393–1405. [CrossRef]
24. Blondel, V.D.; Guillaume, J.-L.; Lambiotte, R.; Lefebvre, E. Fast unfolding of communities in large networks. *J. Stat. Mech. Theory Exp.* **2008**, *2008*, P10008. [CrossRef]
25. Hex-Rays. IDA FLIRT. Available online: https://hex-rays.com/products/ida/tech/flirt/ (accessed on 5 April 2023).
26. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2016), San Diego, CA, USA, 21–24 February 2016.
27. Dullien, T.; Rolles, R. Graph-based comparison of Executable Objects. *Sstic* **2005**, *5*, 3.
28. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable Graph-based Bug Search for Firmware Images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016.
29. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Tan, H.B.K. BinGo: Cross-architecture cross-OS binary search. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016.
30. Wang, S.; Wu, D. In-memory fuzzing for binary code similarity analysis. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, Urbana, IL, USA, 30 October–3 November 2017.
31. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D.X. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In Proceedings of the ACM SIGSAC Conference on Computer Communications Security, Dallas, TX, USA, 30 October–3 November 2017.

32. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 896–899. [CrossRef]
33. Le, Q.V.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the International Conference on Machine Learning, Beijing, China, 21–26 June 2014.
34. Levine; John, R. Linkers and Loaders. *Acm Comput. Surv.* **1999**, *4*, 149–167.
35. Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Lio', P.; Bengio, Y. Graph Attention Networks. *arXiv* **2017**, arXiv:1710.10903.
36. Xie, J.; Girshick, R.B.; Farhadi, A. Unsupervised Deep Embedding for Clustering Analysis. *arXiv* **2016**, arXiv:1511.06335.
37. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* **2014**, arXiv:1412.6980.
38. Reimers, N.; Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *arXiv* **2019**, arXiv:1908.10084.
39. Gu, Y.; Shu, H.; Hu, F. UniASM: Binary Code Similarity Detection without Fine-tuning. *arXiv* **2022**, arXiv:2211.01144.
40. Kuhn, H.W. The Hungarian method for the assignment problem. *Nav. Res. Logist.* **1955**, *2*, 83–97. [CrossRef]
41. Rokon, M.O.F.; Islam, R.; Darki, A.; Papalexakis, E.E.; Faloutsos, M. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), San Sebastian, Spain, 14–16 October 2020; USENIX Association: Berkeley, CA, USA, 2020; pp. 149–163.
42. MalwareBazaar. Available online: https://bazaar.abuse.ch/ (accessed on 5 April 2023).
43. Zynamics BinDiff. Available online: https://www.zynamics.com/bindiff.html (accessed on 5 April 2023).
44. Diaphora-A Free and Open Source Program Diffing Tool. Available online: http://diaphora.re/ (accessed on 5 April 2023).
45. Xu, X.; Fan, M.; Jia, A.; Wang, Y.; Yan, Z.; Zheng, Q.; Liu, T. Revisiting the Challenges and Opportunities in Software Plagiarism Detection. In Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 537–541. [CrossRef]
46. Lin, G.; Wen, S.; Han, Q.L.; Zhang, J.; Xiang, Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
47. Huang, Y.; Shu, H.; Kang, F. DeMal: Module decomposition of malware based on community discovery. *Comput. Secur.* **2022**, *117*, 102680. [CrossRef]
48. Yadegari, B.; Johannesmeyer, B.; Whitely, B.; Debray, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 674–691. [CrossRef]